School of Computation, Information and Technology
Technische Universität München

Lehrstuhl für
Rechnerarchitektur & Parallele Systeme
Prof. Dr. Martin Schulz
Dominic Prinz
Jakob Schäffeler

Lehrstuhl für
Design Automation
Prof. Dr.-Ing. Robert Wille
Stefan Engels

Einführung in die Rechnerarchitektur

Wintersemester 2024/2025

Übungsblatt 2: RISC-V Assembly 28.10.2024 – 01.11.2024

1 RISC-V Simulator Einrichtung

In ERA wird zur Simulation eines RISC-V Prozessors `QtRvSim` – ein Projekt der Tschechischen Technischen Universität – verwendet. Für die erste Einrichtung folgen Sie bitte den folgenden Schritten:

1. Laden Sie sich die passende Installationsdatei für Ihr Betriebssystem herunter: `https://github.com/cvut/qtrvsim/releases/tag/v0.9.8`
   - Ubuntu-User können auch folgendes `PPA` verwenden: `ppa:qtrvsimteam/ppa`
   - Windows-User benutzen die Datei mit `mingw32` im Namen

   oder verwenden Sie die Web-Version (experimentell): `https://comparch.edu.cvut.cz/qtrvsim/app`
2. Belassen Sie die Einstellungen wie sie sind: „No pipeline no cache“ und klicken Sie auf „Example“.
3. In der oberen Hälfte sehen Sie die Register inkl. der zugehörigen Mnemonics und Werte.
4. Links sehen Sie die auszuführenden Instruktionen. Die Instruktionen eines Programms beginnen wie im RISC-V Ökosystem üblich bei Adresse `0x200`.
5. Sie können mithilfe der Reiter „Core“ und „template.S“ zwischen der Prozessor- und Source Code-Ansicht wechseln.
6. Klicken Sie auf „Compile Source and update memory“ (blauer Pfeil nach unten) um den aktuell ausgewählten Source Code zu kompilieren und zu laden.
7. Anschließend können Sie das Programm mit dem Play-Button starten. Als Beispiel wird der Text „Hello world.“ rechts auf dem Terminal ausgegeben.

8. Weitere Beispiele finden Sie hier: `https://gitlab.fel.cvut.cz/b35apo/stud-support/-/tree/master/seminaries/qtrvsim`

9. Tipp: Probieren Sie sich hier aus!

2 Registerbenutzung und (symbolische) Konstanten

a) Machen Sie sich nochmals mit den Registern und deren üblicher Verwendung vertraut. Die Register sind in Tabelle 1 abgebildet.

b) Alle für uns relevanten Befehle sind in Tabelle 2 gelistet. Nutzen Sie möglichst wenige Befehle, um folgende Aufgaben zu bewältigen.

- Laden Sie in das Register `a0` den Wert `0` (sog. Nullen).

```
addi  a0, zero, 0
add   a0, zero, zero
```


- Laden Sie die Konstante `0xF0000000` in das Register `t1`.

```
lui  t1, 0xF0000
```

(12 bit)

- Laden Sie die Konstante `0xABAD1DEA` in das Register `a3`. Verwenden Sie keine Pseudobefehle.

```
addi  a3  0x DEA
      0x FFFF.FDEA

0x ABAD1 DEA  =  y + 0x FFFF.FDEA
y = 0x ABAD1 DEA - (- 0x0000.0216)
  0x ABAD1 DEA +
  0x 0000.0216
  0x ABAD.2000
```

DEA : 1101.1110.1010
1-Komp: 0010.0001.0101
2-Komp: 0010.0001.0110
0x 0000.0216

```
lui   a3, 0x ABAD2
addi  a3, a3, -534

li  a3, 0x ABAD1 DEA
```

3 Einfache Befehle

a) Welchen Wert hat das Register a0 nach der Ausführung des folgenden Programms:

```
main:
    addi a0, zero, -534
    lui a0, 0
    srli a0, a0, 10
    add a0, a0, a0
```

-534 0x FFFF. FDEA
0
0
0

b) Übersetzen Sie die folgenden Terme in RISC-V Assembler. Verwenden Sie dabei nur Befehle aus Tabelle 2.

- $a_0 := a_0 - a_1$

```
sub a0, a0, a1
```

- $a_0 := a_0 + 2^{10}$

```
addi a0, a0, 1024
```

- $a_0 := a_0 + 2^{11}$

```
addi a1, zero, 1
slli a1, a1, 11
add a0, a0, a1
```

- $a_0 := a_0 \cdot 2 + 123$ (vorzeichenlose Multiplikation)

```
slli a0, a0, 1
addi a0, a0, 123
```

- $a_0 := a_0 \cdot 5$ (vorzeichenlose Multiplikation)

```
slli t0, a0, 2
add a0, a0, t0
```

4 Bitmasken und -hacks

Setzen Sie die folgenden Aufgaben um indem Sie Bitmasken und -hacks verwenden.

a) $a_0 = \lfloor a_0/16 \rfloor$ (Ergebnis abgerundet, vorzeichenlose Division)

```
srli a0, a0, 4
```

b) $a_0 = a_0 \mod 256$ (vorzeichenlos)

```
andi a0, a0, 0xFF
```

1.0000.0000

0000 ... 1111.1111

c) Kopieren Sie die unteren 16 Bit von `a1` in die unteren 16 Bit von `a0` und die unteren 16 Bit von `a2` in die oberen 16 Bit von `a0`.

```
slli a1, a1, 16
srli a1, a1, 16
slli a2, a2, 16
or   a0, a1, a2
```

d) Setzen Sie das `a1`-te Bit in `a0` auf Eins. Die restlichen Bits sollen unverändert bleiben.

```
addi t0, zero, 1
sll  t0, t0, a1
or   a0, a0, t0
```

e) Bonus: Setzen Sie das `a1`-te Bit in `a0` auf Null. Die restlichen Bits sollen unverändert bleiben.

```
addi t0, zero, 1
sll  t0, t0, a1
or   a0, a0, t0
xor  a0, a0, t0
```

5 Vorüberlegungen zur Hausaufgabe

a) Was passiert, wenn man die in der Zentralübung erwähnte Formel zur Berechnung des Werts einer Binärzahl ($a = \sum_{i=0}^{n} a_i \cdot 2^i$) auf negative Indizes erweitert?

b) Welchen Wertebereich kann man mit 8 binären Vorkommastellen (ohne Vorzeichen) und 8 binären Nachkommastellen (8.8) erreichen?

Vorkomma: $2^8 - 1 = 255$

Nachkomma: $2^{-8} = \frac{1}{2^8} = \frac{1}{256} = 0{,}000\,39\ldots$

$\frac{255}{256} = 1 - 2^{-8} = 0{,}996\,037\ldots$

Wertebereich $[0,\ 255{,}996\,037\ldots]$

c) Wieviele Bit bräuchte man mindestens, um Zahlen von 0 bis 100 darzustellen, sodass der Abstand zwischen zwei darstellbaren Werten maximal 0.005 beträgt?

	Kombinationen	Bit
0 – 1	2	1
0 – 3	4	2
0 – 7	8	3

Vorkommastellen $\lceil \log_2(100) \rceil = 7$ bit

Nachkommastellen $\lceil -\log_2(0.005) \rceil = 8$ bit

d) Wie sieht die Addition bzw. Subtraktion in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?

e) Wie sieht die Multiplikation in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?

8.8

1,5 = 0000.0001, 1000.0000 8 8

3,25 = 0000.0011, 0100.0000

Multiplikation: 0000.0100, 1110.0000.0000.0000

4, 875

f) Wie sieht die Zahl π im 8.8-Format aus? Was muss man beachten?

$\lfloor 3{,}14 \cdot 2^8 \rfloor = 804$

$\frac{804}{2^8} \approx 3{,}14\ldots$

g) Wie kann man beliebige Zahlen in Festkommazahlen umwandeln?

siehe f)

6 Festkommarechnung (Hausaufgabe 02)

Bearbeitung und Abgabe der Hausaufgabe 02 auf `https://artemis.in.tum.de/courses/401` bis **Sonntag, den 03.11.2024, 23:59 Uhr**.

7 Referenzmaterial

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments/Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3–6	x28–31	Temporary registers

Abbildung 1: RISC-V 32-Bit Register

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0= rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0= rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 − rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

*Encoded in $instr_{31:25}$, the upper seven bits of the immediate field

Abbildung 2: RISC-V 32-Bit Integerbefehle

